Procedia Computer Science

Volume 80, 2016, Pages 201–211

ICCS 2016. The International Conference on Computational Science

A Case Study in Adjoint Sensitivity Analysis of Parameter Calibration

Johannes Lotz[1], Marc Schwalbach[2], and Uwe Naumann[1]

[1] LuFG Informatik 12: Software and Tools for Computational Engineering, RWTH Aachen, Germany
[lotz, naumann]@stce.rwth-aachen.de

[2] Von Karman Institute for Fluid Dynamics, Belgium
marc.schwalbach@vki.ac.be

Abstract

Adjoint sensitivity computation of parameter estimation problems is a widely used technique in the field of computational science and engineering for retrieving derivatives of a cost functional with respect to parameters efficiently. Those derivatives can be used, e.g. for sensitivity analysis, optimization, or robustness analysis. Deriving and implementing adjoint code is an error-prone, non-trivial task which can be avoided by using Algorithmic Differentiation (AD) software. Generating adjoint code by AD software has the downside of usually requiring a huge amount of memory as well as a non-optimal run time. In this article, we couple two approaches for achieving both, a robust and efficient adjoint code: symbolically derived adjoint formulations and AD. Comparisons are carried out for a real-world case study originating from the remote atmospheric sensing simulation software JURASSIC developed at the Institute of Energy and Climate Research – Stratosphere, Research Center Jülich. We show, that the coupled approach outperforms the fully algorithmic approach by AD in terms of run time and memory requirement and argue that this can be achieved while still preserving the desireable feature of AD being automatic.

Keywords: Adjoints, Algorithmic Differentiation, Optimization, C++

1 Problem Statement and Summary of Results

In this article a comparison is carried out between different approaches for computing first-order adjoint sensitivities of a parameter calibration problem. We consider the application of adjoint Algorithmic Differentiation [6, 14] (also known as Automatic Differentiation; AD) as well as a combination of AD with symbolically derived adjoint formulations for the respective optimizer of the parameter calibration problem. The latter strategy was already successfully pursued for various algorithms, see for example [9, 16, 17]. In particular [16] derives a combined symbolic adjoint formulation as described in this article for a general nonlinear system solver. We apply these results to the parameter calibration problem and carry out a case study showing

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.310

the benefits. The sensitivities computed by the adjoint code are of great value when solving e.g. bilevel optimization problems [3, 13] targeted in the BeProMod project[1], or in general for analyzing the robustness of the optimal solution [1] or for uncertainty quantification [12].

Adjoint AD is a semantic program transformation that *automatically* generates a program which computes the gradient of a target functional at a cost that is a constant multiple of the cost of an evaluation of the target functional itself. In particular, that means the gradient is computed at a cost independent of the number of parameters. This advantage has the downside of requiring a huge amount of additional memory, since a data flow reversal is required, i.e. all intermediate variables need to be accessed in reverse order. Coupling adjoint AD with symbolic adjoint formulations can resolve this issue of high memory consumption and in addition possibly reduces the computational complexity significantly. Such a coupling can be implemented seamlessly with modern AD tools and preserve the desireable feature of AD being automatic and efficient. In this article we consider the optimization problem isolated from how it is possibly embedded in a surrounding overall simulation program. An analogue embedding is explained in more detail in the articles cited above, in particular [16].

The parameter calibration problem is stated as follows. For given observations $\mathbf{o} \in \mathbb{R}^m$, known parameters $\boldsymbol{\lambda} \in \mathbb{R}^m$, and forward model $g(\mathbf{x}, \boldsymbol{\lambda}_i)$, unknown model parameters $\mathbf{x} \in \mathbb{R}^n$ are sought such that for the residual function

$$F = [\mathbf{o}_i - g(\mathbf{x}, \boldsymbol{\lambda}_i)]_{i=1\ldots m} \tag{1}$$

the least squares cost function

$$G(\mathbf{x}, \boldsymbol{\lambda}) = F^T F = \sum_{i=1}^{m} [\mathbf{o}_i - g(\mathbf{x}, \boldsymbol{\lambda}_i)]^2 \tag{2}$$

is minimized, yielding an overall problem statement

$$\hat{\mathbf{x}}(\boldsymbol{\lambda}) = S(\mathbf{x}_0, \boldsymbol{\lambda}) = \arg\min_{\mathbf{x} \in \mathbb{R}^n} G(\mathbf{x}, \boldsymbol{\lambda}) \quad \text{with optimum} \quad G(\hat{\mathbf{x}}(\boldsymbol{\lambda}), \boldsymbol{\lambda}), \tag{3}$$

where S denotes the optimization method (Gauss-Newton [5] in our case) and $\mathbf{x}_0$ the starting value. In addition to calibrating the unknown parameters $\mathbf{x} \in \mathbb{R}^n$, we are also interested in first-order sensitivities of some scalar target functional $J(\hat{\mathbf{x}}(\boldsymbol{\lambda}))$ with respect to the input parameters $\boldsymbol{\lambda}$, i.e. the gradient $d_{\boldsymbol{\lambda}} J \in \mathbb{R}^m$.

After a short introduction into the used notation and to AD and its capabilities in Section 2, we present two different approaches for the computation of first-order sensitivities in Section 3. Besides a fully algorithmic approach by plain application of AD to the solver implementation, a coupled approach is proposed taking mathematical properties into account and symbolically deriving adjoint formulations for decreasing memory footprint and increasing performance. In addition, we address the issue of efficiently computing the Jacobian of the residual F required during the minimization of G. Since this Jacobian computation is also part of the overall optimization algorithm S, computing its sensitivities implicitly involves higher-order derivatives of F. We go into further details there and recapitulate results from [11] concerning the computation of $d_{\mathbf{x}} F$. The theoretically deduced memory and run time benefit is underpinned by a case study coming from a real-life atmospheric remote sensing application developed at the Research Center Juelich in Section 4. We close with conclusion and outlook in Section 5.

[1] see Acknowledgments

2 Algorithmic Differentiation

This section introduces the used notation and gives a brief overview of AD [6, 14]. Scalars are non-bold lower case letters, vectors are bold lower case letters, and matrices are non-bold upper case letters. Total derivatives of outputs $\mathbf{y} \in \mathbb{R}^m$ with respect to inputs $\mathbf{x} \in \mathbb{R}^n$, i.e. the Jacobian, are denoted by $d_{\mathbf{x}}\mathbf{y} \in \mathbb{R}^{m \times n}$. Second derivatives are correspondingly denoted by $d_{\mathbf{xx}}\mathbf{y}$. Similarly, partial derivatives of $\mathbf{y}$ with respect to $\mathbf{x}$ are denoted by $\partial_{\mathbf{x}}\mathbf{y}$. Particular elements of a vector or a matrix are denoted by indexes, e.g. $\mathbf{x}_i$ for the i-th element of a vector $\mathbf{x} \in \mathbb{R}^n$.

Without loss of generality, for a given implementation of the multi-variate twice continuously differentiable function

$$y = f(\mathbf{x}), \quad f : \mathbb{R}^n \to \mathbb{R}, \quad \text{with} \quad \mathbf{x} \in \mathbb{R}^n, \text{ and } y \in \mathbb{R}\,, \tag{4}$$

AD is a semantical program transformation which automatically generates programs computing sensitivities. This transformation is usually done under support of an AD tool. Which tool to use depends strongly on the programming language as well as the specific application. Generally, AD provides two different models. Firstly, the *tangent* or *forward* model of $f(\mathbf{x})$ is given as

$$y^{(1)} = d_{\mathbf{x}} f(\mathbf{x}) \cdot \mathbf{x}^{(1)}\,, \tag{5}$$

with input tangents $\mathbf{x}^{(1)} \in \mathbb{R}^n$ and output tangent $y^{(1)} \in \mathbb{R}$. Using AD, one evaluation of the tangent model can be performed at a cost of $\mathcal{O}(1) \cdot \text{cost}(f)$. For computing all individual gradient entries $[d_{\mathbf{x}} f(\mathbf{x})]_i$, n inner vector products need to be computed with $\mathbf{x}^{(1)}$ ranging over the Cartesian basis vectors in $\mathbb{R}^n$ yielding an accumulated cost of $\mathcal{O}(n) \cdot \text{cost}(f)$. Secondly, the *adjoint* or *reverse* model of $f(\mathbf{x})$ is given as

$$\mathbf{x}_{(1)} = d_{\mathbf{x}} f(\mathbf{x})^T \cdot y_{(1)}\,, \tag{6}$$

with output adjoint $y_{(1)} \in \mathbb{R}$ and input adjoints $\mathbf{x}_{(1)} \in \mathbb{R}^n$. One evaluation of the adjoint model can also be performed at a cost of $\mathcal{O}(1) \cdot \text{cost}(f)$ (usually with a bigger constant than for the tangent case). In contrast to the tangent model, all indiviual gradient entries can be computed in *one* evaluation yielding a huge run time benefit. The implementation of adjoint mode AD requires a data flow reversal, where required [7] intermediate variables need to be accessible in reverse order. This requires a stack-like data structure, that is filled during a forward run and used during the reverse run. The memory consumption is therefore one of the biggest challenges for adjoint mode AD. Based on the tangent and the adjoint model, recursive application generates higher derivative models. In particular applying tangent differentiation to the adjoint model yields the so-called tangent-over-adjoint model

$$\mathbf{x}^{(2)}_{(1)} = d_{\mathbf{x}} f(\mathbf{x})^T \cdot y^{(2)}_{(1)} + y_{(1)} \cdot d_{\mathbf{xx}} f(\mathbf{x}) \cdot \mathbf{x}^{(2)}\,, \tag{7}$$

with adjoint $y_{(1)} \in \mathbb{R}$, tangents $\mathbf{x}^{(2)} \in \mathbb{R}^n$, and second-order adjoints $y^{(2)}_{(1)} \in \mathbb{R}$ and $\mathbf{x}^{(2)}_{(1)} \in \mathbb{R}^n$. The Hessian $d_{\mathbf{xx}} f(\mathbf{x}) \in \mathbb{R}^{n \times n}$ can be computed at a cost of $\mathcal{O}(n) \cdot \text{cost}(f)$ by performing n runs of the above given model with $y_{(1)} = 1$, $y^{(2)}_{(1)} = 0$, and $\mathbf{x}^{(2)}$ ranging over the Cartesian basis vectors in $\mathbb{R}^n$. Each individual evaluation of the model computes one row or column of the Hessian. Analogously, adjoint-over-tangent, tangent-over-tangent and adjoint-over-adjoint models can be used to compute the Hessian, see [14]. Multiple software packages for the implementation of AD are listed on `www.autodiff.org`. The code for this project is written in C++ and our C++ operator overloading library `dco/c++` [10, 9, 15, 18] is used. For more articles related to AD, the interested reader is referred to [2, 4].

3 Adjoint Sensitivity Analysis

In the following, we first briefly recapitulate the algorithmic results from [11] for the computation of the Jacobian $d_{\mathbf{x}}F$ followed by the theoretical description of the two approaches for computing first-order adjoints, i.e. sensitities of the target functional. Measurements of memory consumption and run times for the different approaches are shown in Section 4.

3.1 Computing the Jacobian

The Gauss-Newton method requires the Jacobian $d_{\mathbf{x}}F$ of the residual F, see Equation (1). Computing the entire Jacobian in adjoint mode AD would involve m adjoint model evaluations of F, yielding a total cost of $\mathcal{O}(m) \cdot \text{cost}(F)$. Since usually $m > n$, this is presumably less efficient than computing the Jacobian using tangent mode AD, which involves n tangent model evaluations at a total cost of $\mathcal{O}(n) \cdot \text{cost}(F)$. However, note that mutual independence of the individual residual elements

$$\mathbf{y}_i = \mathbf{o}_i - g(\mathbf{x}, \boldsymbol{\lambda}_i) \tag{8}$$

can be exploited. This corresponds to an ensemble structure [6, 11, 19] of which we take advantage by expanding the vector of the parameters $\mathbf{x} \in \mathbb{R}^n$ into a matrix $X \in \mathbb{R}^{m \times n}$ with rows $X^i = \mathbf{x}^T$ for $i = 1, ..., m$. We now define the extended residual function

$$\hat{\mathbf{y}} = \hat{F}(X, \boldsymbol{\lambda}) = \left[\mathbf{o}_i - g(X^i, \boldsymbol{\lambda}_i)\right]_{i=1...m} \text{ with } \hat{\mathbf{y}} \in \mathbb{R}^m \tag{9}$$

which fulfills $\hat{F}(X, \boldsymbol{\lambda}) \equiv F(\mathbf{x}, \boldsymbol{\lambda})$. The adjoint model of $\hat{F}$ with respect to X is given as

$$X^i_{(1)} = \left[d_{X^i}\hat{F}\right]^T \cdot \hat{\mathbf{y}}_{(1)} \overset{(*)}{=} d_{X^i}\hat{F}_i = d_{\mathbf{x}}F_i \; . \tag{10}$$

Equality $(*)$ holds, since the i-th residual only depends on the i-th row of the complete matrix X. The Jacobian can now be obtained at a cost of $\mathcal{O}(1) \cdot \text{cost}(\hat{F}) = \mathcal{O}(1) \cdot \text{cost}(F)$ by setting the adjoint $\hat{\mathbf{y}}_{(1)} = \mathbf{1} = (1, ..., 1)^T$. After the adjoint model evaluation we get $X_{(1)} = d_{\mathbf{x}}F$. In addition, the m individual adjoint computations of $X^i_{(1)}$ can be run in parallel using multiple processes, where each thread computes one row of the Jacobian, e.g. by using OpenMP. Memory and run time measurements are shown in Section 4.

3.2 First-Order Sensitivities of Optimizer

First-order sensitivities are computed in two different ways. First, by plain application of adjoint mode AD to the implementation of the target functional J including optimizer S. Secondly, by exploiting the first-order optimality condition for $\hat{\mathbf{x}}$ to derive a sensitivity equation via symbolic differentiation of G. The former approach is called *algorithmic approach* and the latter *symbolic approach* in the following. As we will see later, the symbolic approach also involves some algorithmic adjoint computations of the underlying function G but completely avoids algorithmic adjoints of the optimizer algorithm S.

We consider the adjoint model

$$\boldsymbol{\lambda}_{(1)} = \left[d_{\boldsymbol{\lambda}}J\right]^T \cdot J_{(1)} = \left[\partial_{\boldsymbol{\lambda}}J\right]^T \cdot J_{(1)} + \left[\partial_{\boldsymbol{\lambda}}\hat{\mathbf{x}}\right]^T \cdot \underbrace{\left[\partial_{\hat{\mathbf{x}}}J\right]^T \cdot J_{(1)}}_{\hat{\mathbf{x}}_{(1)}} \; . \tag{11}$$

Algorithmic Approach Setting $J_{(1)} = 1$ will allow us to compute the first-order sensitivities with a single adjoint run of J including S at a cost of $\mathcal{O}(1) \cdot (\text{cost}(S) + \text{cost}(J))$. However, note that the adjoint evaluation of $S(\mathbf{x}_0, \boldsymbol{\lambda})$ involves an adjoint evaluation of the Jacobian calculation $d_{\mathbf{x}}F$. Since the Jacobian is computed already using adjoint mode AD, this implicitly requires

adjoint-over-adjoint computations, which usually yields less efficient second-order adjoint code than tanget-over-adjoint mode[2] [6, 14]. This can be avoided by exploiting the symmetry of the Hessian and using the tangent-over-adjoint mode instead, yielding less memory requirements and shorter run times. Measurements are shown in Section 4.

Symbolic Approach The first-order sensitivities $d_{\boldsymbol{\lambda}} J$ can also be computed using symbolic differentiation of the first-order optimality condition

$$d_{\mathbf{x}} G\left(\mathbf{x}, \boldsymbol{\lambda}\right) = \mathbf{0}\ . \tag{12}$$

Differentiating this equation at $\mathbf{x} = \hat{\mathbf{x}}$ with respect to $\boldsymbol{\lambda}$ using the implicit function theorem taking the dependency $\hat{\mathbf{x}}(\boldsymbol{\lambda})$ into account yields

$$\partial_{\mathbf{x}\boldsymbol{\lambda}} G + \partial_{\mathbf{x}\mathbf{x}} G \cdot \partial_{\boldsymbol{\lambda}} \mathbf{x} = \mathbf{0}. \tag{13}$$

Using the transpose of Equation (13) in the last term of Equation (11) yields

$$\boldsymbol{\lambda}_{(1)} = \left[\partial_{\boldsymbol{\lambda}} J\right]^T \cdot J_{(1)} - \left[\partial_{\mathbf{x}\boldsymbol{\lambda}} G\right]^T \cdot \underbrace{\left[\partial_{\mathbf{x}\mathbf{x}} G\right]^{-1} \cdot \hat{\mathbf{x}}_{(1)}}_{=:\mathbf{z}}\ , \tag{14}$$

where the transposal of $\partial_{\mathbf{x}\mathbf{x}} G$ is neglected since the Hessian is symmetric. The first term on the right is computed as an algorithmic adjoint projection of J only, i.e. without evaluating S (because the partial derivative is required), at a cost of $\mathcal{O}(1) \cdot \mathrm{cost}(J)$. The second term results in solving the linear system

$$\partial_{\mathbf{x}\mathbf{x}} G \cdot \mathbf{z} = \hat{\mathbf{x}}_{(1)} \tag{15}$$

for $\mathbf{z}$. $\partial_{\mathbf{x}\mathbf{x}} G$ can be computed by using second-order tangent-over-adjoint mode of G at $\mathbf{x} = \hat{\mathbf{x}}$ at a cost of $\mathcal{O}(n) \cdot \mathrm{cost}(G)$. The linear system could also be solved with an approximated Hessian matrix. This especially seems obvious, when using a Gauss-Newton method to solve the parameter calibration problem as done in the case study in Section 4. Once the system has been solved for $\mathbf{z}$, we can compute

$$-\left[\partial_{\mathbf{x}\boldsymbol{\lambda}} G\right]^T \cdot \left[\partial_{\mathbf{x}\mathbf{x}} G\right]^{-1} \cdot \hat{\mathbf{x}}_{(1)} = -\left[\partial_{\mathbf{x}\boldsymbol{\lambda}} G\right]^T \cdot \mathbf{z} \tag{16}$$

from Equation (14) with a single call of the second-order adjoint of G at $\mathbf{x} = \hat{\mathbf{x}}$ at a cost of $\mathcal{O}(1) \cdot \mathrm{cost}(G)$. This is performed by one evaluation of the tangent-over-adjoint model

$$\begin{pmatrix} \mathbf{x}_{(1)}^{(2)} \\ \boldsymbol{\lambda}_{(1)}^{(2)} \end{pmatrix} = \left[\frac{\partial G}{\partial(\mathbf{x}, \boldsymbol{\lambda})}\right]^T \cdot G_{(1)}^{(2)} + G_{(1)} \cdot \frac{\partial^2 G}{\partial(\mathbf{x}, \boldsymbol{\lambda})^2} \cdot \begin{pmatrix} \mathbf{x}^{(2)} \\ \boldsymbol{\lambda}^{(2)} \end{pmatrix} \tag{17}$$

and setting $G_{(1)} = 1$, $G_{(1)}^{(2)} = 0$, $\boldsymbol{\lambda}^{(2)} = \mathbf{0}$, and $\mathbf{x}^{(2)} = -\mathbf{z}$. The required matrix-vector product from Equation (16) is returned in $\boldsymbol{\lambda}_{(1)}^{(2)}$. Note that $\mathrm{cost}(G) = \mathcal{O}(1) \cdot \mathrm{cost}(F)$ and the optimizer S has to run *only once* at a cost of $\mathrm{cost}(S)$ to obtain $\hat{\mathbf{x}}$ at the beginning. This way the symbolic approach avoids the tool-based data flow reversal of S as well as the problem of generating and running the reverse-over-reverse model of F. This yields a huge reduction in memory requirement and a moderate improvement in terms of run time as shown later in the case study Section 4.

4 Case Study

In this section, we consider the different approaches presented in the previous section applied to a real-life related problem. The model used for these tests is based on the *Jülich Rapid Spec-*

[2]Note, that this statement is tool and application dependent. If enough memory is available, we've seen cases where `dco/c++` computes the Hessian more efficiently in adjoint-over-adjoint mode.

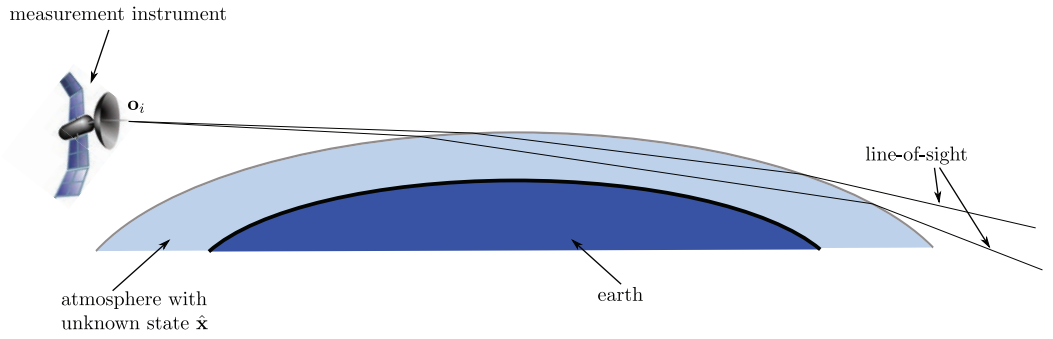

Figure 1: Overview of the model setup for the case study. For measurement data $\mathbf{o}_i$, JURASSIC2 solves for the unknown atmospheric state $\hat{\mathbf{x}}$ using an inverse problem formulation [11].

tral Simulation Code Version 2 (JURASSIC2) which is developed by the Institute of Energy and Climate Research – Stratosphere, Research Center Jülich [20] and was already successfully combined with AD [11]. The code used in this section is a simplified version of this model and was kindly provided by Jörn Ungermann[3]. It is used in the field of atmospheric remote sensing and solves an inverse problem to compute an approximation of the 3-dimensional space-filling atmospheric state in the upper troposphere and lower stratosphere for given radiance measurements. Measurements are performed by a measurement instrument mounted on an airplain [8] or a satellite (not yet realized). An overview of the measurement setup is shown in Fig. 1. The numerics for solving the inverse problem are based on a forward model simulating radiances from a guessed atmospheric state and line-of-sight elevations, two of which are sketched in the figure. With the symbols from the introduction, the forward model is given as $\mathbf{y}_i = g(\mathbf{x}, \boldsymbol{\lambda}_i)$ with $\mathbf{y} \in I\!R^m$ denoting the simulated radiances, $\mathbf{x} \in I\!R^n$ the atmospheric parameters, and $\boldsymbol{\lambda} \in I\!R^m$ the line-of-sight elevations. For given measured radiances $\mathbf{o} \in I\!R^m$, the residual vector $F \in I\!R^m$ is computed as

$$F = (\mathbf{o}_i - g(\mathbf{x}, \boldsymbol{\lambda}_i))_{i=1\ldots m} \; . \tag{18}$$

For this case study, observations are generated by a twin experiment, which is a widespread technique for benchmarking inverse problem algorithms. Artificial measurements are obtained by running the forward model with a known set of parameters $\mathbf{x}^s$. The cost functional consists of the sum of the squares of the residuals already shown in the introduction plus a regularization term since the inverse problem is ill-posed. The cost function is then given by

$$G\,(\mathbf{x}, \boldsymbol{\lambda}) = F^T S_\epsilon^{-1} F + (\mathbf{x} - \mathbf{x}_a)^T S_a^{-1} (\mathbf{x} - \mathbf{x}_a) \; , \tag{19}$$

where $S_\epsilon \in I\!R^{m \times m}$ is a measurement error correlation matrix, $\mathbf{x}_a \in I\!R^n$ denotes typical atmospheric values taken from historic data, and $S_a \in I\!R^{n \times n}$ is a Tikhonov regularization matrix, here $S_a = S_\epsilon = I$ and $\mathbf{x}_a$ is chosen to be a constant value identical to the starting value for the Gauss-Newton method $\mathbf{x}_0$. The inverse problem is to solve

$$\hat{\mathbf{x}} = S(\mathbf{x}_0, \boldsymbol{\lambda}) = \arg \min_{\mathbf{x} \in I\!R^n} G\,(\mathbf{x}, \boldsymbol{\lambda}) \; . \tag{20}$$

Applying Gauss-Newton, i.e. neglecting second-order derivatives of F when computing $d_{\mathbf{xx}} G$ in Newton's method, yields the iteration formula

$$\mathbf{x}_{i+1} = \mathbf{x}_i - (S_a^{-1} + d_{\mathbf{x}} F^T S_\epsilon^{-1} d_{\mathbf{x}} F)^{-1} \cdot \left(S_a^{-1} (\mathbf{x}_i - \mathbf{x}_a) + d_{\mathbf{x}} F^T S_\epsilon^{-1} F \right) \; . \tag{21}$$

[3]Institute of Energy and Climate Research, Research Center Jülich

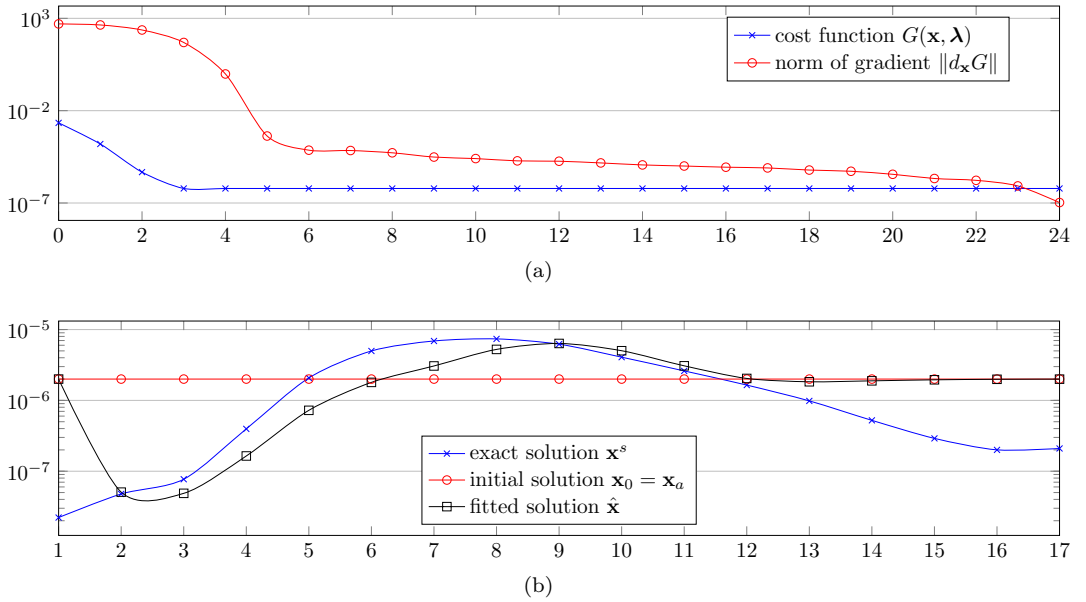

Figure 2: Convergence behavior of the Gauss-Newton method applied to the parameter calibration problem Equation (20) is shown in (a). Corresponding exact, initial, and fitted solution after convergence is shown in (b).

The convergence behavior as well as the fitted solution is shown in Fig. 2. As convergence criterion we chose the two norm of the gradient $\|d_{\mathbf{x}}G\|_2$. As shown in Figure (a), the cost function as well as the norm of the gradient decrease very quickly at the beginning until iteration 6. After that, the cost function stays quite constant while the norm of the gradient continues decreasing down to 10^{-7} in iteration 24. As shown in Figure (b), the initial solution $\mathbf{x}_0$ is set to $2.0 \cdot 10^{-6}$. The fitted solution $\hat{\mathbf{x}}$ is a good approximation to the exact solution $\mathbf{x}^s$ inbetween indexes 2 and 12. The model does not return any information for the remaining indexes, since the fitted solution is quite identical to the reference state $\mathbf{x}_a$ used in the regularization term in Equation (21).

Concerning derivative computations, let's first consider run time and memory consumption of the two different approaches to the computation of the Jacobian matrix $d_{\mathbf{x}}F$ required in Equation (21) shown in Fig. 3(a). As expected, the run time of plain application of adjoint mode AD for computing $d_{\mathbf{x}}F$ at a cost of $\mathcal{O}(m) \cdot \text{cost}(F)$ is much higher compared to the approach exploiting the mutual independence of the computations of each F_i. Both, run time as well as memory consumption have a lower order of complexity as can be seen in the double-logarithmic plot. The memory consumption of the version exploiting the ensemble structure seems to be constant, which is due to the fact that the additional memory consumption is even below the base memory allocation done by the program executable anyway. Figure 3(b) shows run time and memory consumption for a fully algorithmic approach to the computation of sensitivities $d_{\boldsymbol{\lambda}}J$, once with the implicitly required reverse-over-reverse mode during the computation of the Jacobian and once with making use of a forward-over-reverse mode locally instead. As can be seen, the complexity classes seem to be the same for both, run time as well as memory consumption (similar slopes). Nonetheless, a non-negligible offset can be identified especially for the memory.

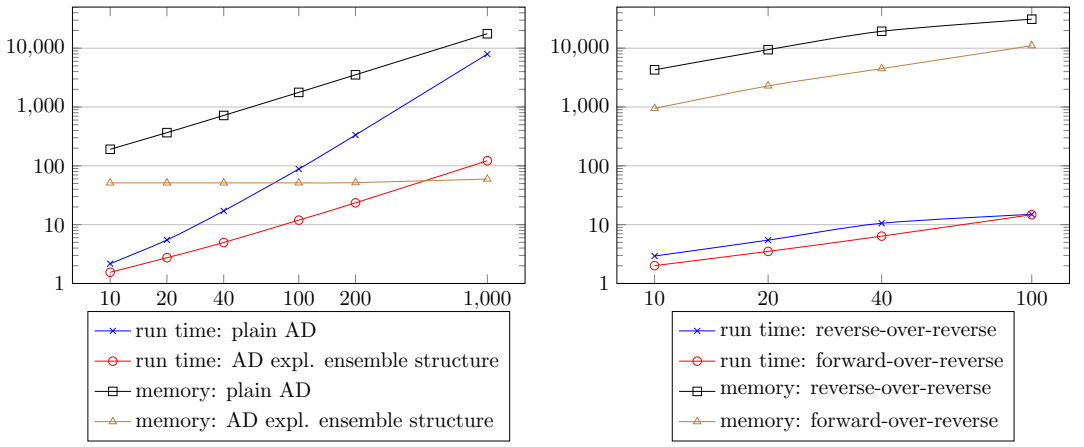

Figure 3: Overall run time in seconds and memory consumption in megabytes for the different approaches to the computation of the Jacobian is shown in (a). The impact of the chosen second-order differentiation method (reverse-over-reverse and forward-over-reverse) on the computation of sensitivities using the algorithmic approach is shown in (b).

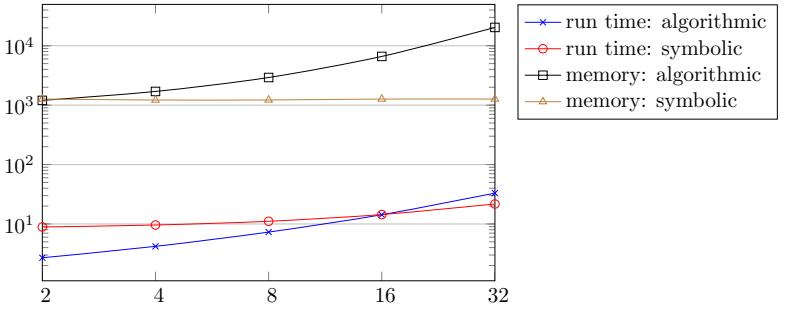

Figure 4: Run time in seconds and memory consumption in megabytes for algorithmic and symbolic approaches to the computation of sensitivities $d_{\boldsymbol{\lambda}}J$ and $d_{\mathbf{o}}J$ over a varying number of Gauss-Newton iterations.

Fig. 4 shows run time and memory consumption of algorithmic and symbolic approaches to the computation of the sensitivities of J with respect to the parameters. The plot in double-logarithmic scale shows that the run time of the symbolic approach lives in a lower-order complexity class with respect to the number of required iterations for convergence. The break-even point is in this case at 16 iterations, above which the symbolic approach is faster. Memory-wise, the symbolic approach outperforms the algorithmic by far, since no data flow reversal (see Section 2) is required for the iteration process itself. The memory consumption therefore only depends on the dimension of $\mathbf{x}$ (since $d_{\mathbf{xx}}G$ needs to be saved somewhere) and is independent of the number of performed iterations.

As already mentioned in Section 3, the algorithmic approach computes (similar to a finite difference approximation) sensitivities of what is actually computed. The symbolic approach on the other hand requires full convergence of the nonlinear solver since the adjoint sensitivity

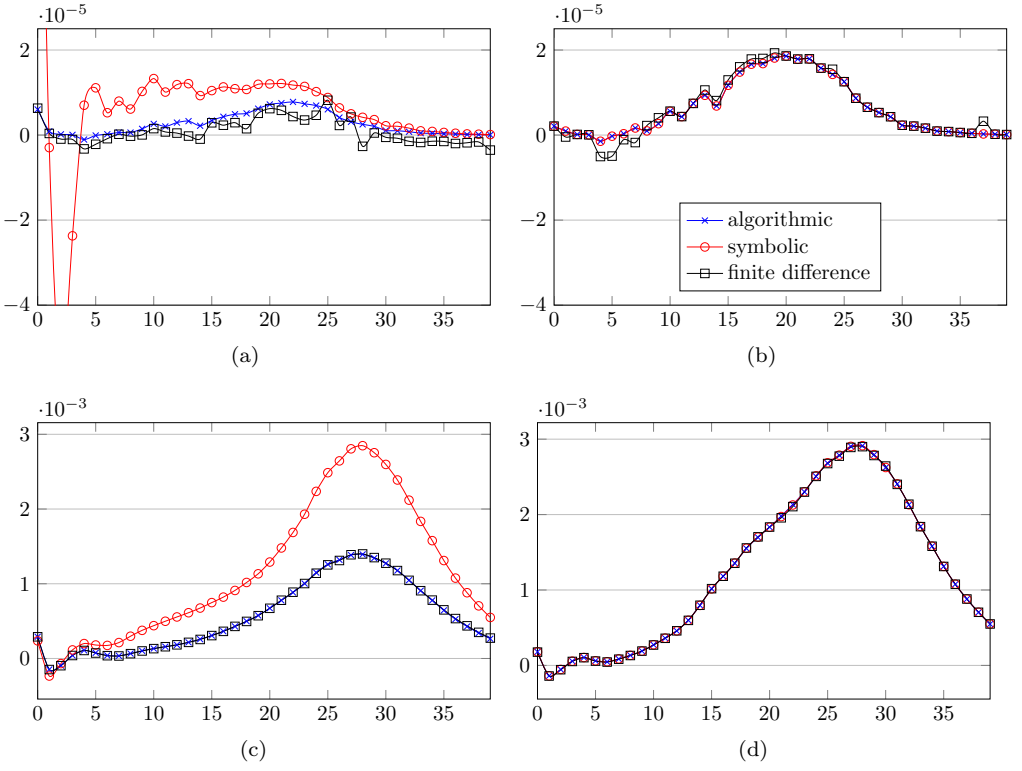

Figure 5: Sensitivities computed by algorithmic mode, symbolic mode, and by finite differences for the parameters $\boldsymbol{\lambda}$ (top) and the observations $\mathbf{o}$ (bottom) after one Gauss-Newton iteration (left) and after convergence (right).

equations are derived under the assumption of $d_{\mathbf{x}}G$ being exactly 0 (see e.g. [6, Ch. 15] for further reading). This is visualized in Fig. 5. The sensitivities $d_{\boldsymbol{\lambda}}J$ and $d_{\mathbf{o}}J$ are computed after 2 Gauss-Newton iterations (i.e. not converged state) and after 24 iterations (i.e. converged state, see Fig. 2(a)). We compare the algorithmic approach, the symbolic approach, and a finite difference approximation. Figures 5(a) and 5(c) clearly show that algorithmic and symbolic approaches result in different values. The finite difference approximation behaves quite similar to the algorithmic approach which is as expected. On the other hand, Figures 5(b) and 5(d) nicely show that all approaches converge to the same sensitivities when solving the nonlinear system with good accuracy.

5 Conclusion and Outlook

On the one hand, the case study shows that a coupled approach is possibly advantageous over a purely algorithmic approach. The accuracy of the computed adjoint sensitivities on the other hand makes it desireable to have a fully algorithmic approach available for verification reasons. Further steps should include an analysis of iterative linear solvers that possibly could be used within the Gauss-Newton iteration as well as the effect of using only the approximate Hessian for the sensitivity computation in the symbolic approach. In addition, the results shown here

for the case study should be applied to the problems occuring in the BeProMod project, which would also include exploration of parallelization possibilities.

Acknowledgments

This work was financially supported by the BeProMod project, which is part of the NRW-Strategieprojekt BioSC funded by the Ministry of Innovation, Science and Research of the German State of North Rhine-Westphalia. In addition, we thank Jörn Ungermann for providing the stripped-down version of JURASSIC2 for the case study.

References

[1] M. Beckers and U. Naumann. Uncertainty Quantification for First-Order Nonlinear Optimization Algorithms. In *CFD & Optimization*. ECCOMAS Thematic Conference, 2011.

[2] C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors. *Advances in Automatic Differentiation*, volume 64 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2008.

[3] G. M. Bollas, P. I. Barton, and A. Mitsos. Bilevel optimization formulation for parameter estimation in vapor–liquid (–liquid) phase equilibrium problems. *Chemical Engineering Science*, 64(8):1768–1783, 2009.

[4] S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, editors. *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2012.

[5] S. Gratton, A. S. Lawless, and N. K. Nichols. Approximate Gauss-Newton methods for nonlinear least squares problems. *SIAM Journal on Optimization*, 18(1):106–132, 2007.

[6] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[7] L. Hascoët, U. Naumann, and V. Pascual. To-be-recorded analysis in reverse mode automatic differentiation. *Future Generation Computer Systems*, 21:1401–1417, 2005.

[8] M. Kaufmann, J. Blank, T. Guggenmoser, J. Ungermann, A. Engel, M. Ern, F. Friedl-Vallon, D. Gerber, J.-U. Grooß, G. Günther, et al. Retrieval of three-dimensional small scale structures in upper tropospheric/lower stratospheric composition as measured by GLORIA. *Atmospheric measurement techniques discussions*, 7:4229–4274, 2014.

[9] J. Lotz, U. Naumann, R. Hannemann-Tamás, T. Ploch, and A. Mitsos. Higher-order discrete adjoint ODE solver in C++ for dynamic optimization. *Procedia Computer Science*, 51:256 – 265, 2015.

[10] J. Lotz, U. Naumann, M. Sagebaum, and M. Schanen. Discrete adjoints of PETSc through dco/c++ and adjoint MPI. *Euro-Par 2013 Parallel Processing*, pages 497–507, 2013.

[11] J. Lotz, U. Naumann, and J. Ungermann. Hierarchical algorithmic differentiation: A case study. In *[4]*, pages 187–196. Springer, 2012.

[12] O. Le Maître. *Spectral Methods of Uncertainty Quantification: With Applications to Computational Fluid Dynamics*. Scientific Computing. Springer, 2nd edition, 2010.

[13] A. Mitsos, B. Chachuat, and P. I. Barton. Towards global bilevel dynamic optimization. *Journal of Global Optimization*, 45(1):63–93, 2009.

[14] U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation*. Number 24 in Software, Environments, and Tools. SIAM, Philadelphia, PA, 2012.

[15] U. Naumann, K. Leppkes, and J. Lotz. `dco/c++` user guide. Technical Report AIB-2014-03, RWTH Aachen University, January 2014.

[16] U. Naumann, J. Lotz, K. Leppkes, and M. Towara. Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of nonlinear equations. *ACM Trans. Math. Softw.*, 41(4):26:1–26:21, October 2015.

[17] N. Safiran, J. Lotz, and U. Naumann. Second-order tangent solvers for systems of parameterized nonlinear equations. *Procedia Computer Science*, 51:231 – 238, 2015.

[18] M. Sagebaum, N. R. Gauger, U. Naumann, J. Lotz, and K. Leppkes. Algorithmic differentiation of a complex C++ code with underlying libraries. *Procedia Computer Science*, 18:208–217, 2013.

[19] M. Schwalbach. Adjoint algorithmic differentiation for large-scale data assimilation. Master's thesis, RWTH Aachen University, 2015.

[20] J. Ungermann, L. Hoffmann, P. Preusse, M. Kaufmann, and M. Riese. Tomographic retrieval approach for mesoscale gravity wave observations by the premier infrared limb-sounder. *Atmospheric Measurement Techniques*, 3(2):339–354, 2010.